On the application of SAT solvers to the Test Suite Minimization Problem

Franco Arito, Francisco Chicano, and Enrique Alba

Departamento de Lenguajes y Ciencias de la Computación,
University of Málaga, Spain
{franco,chicano,eat}@lcc.uma.es

Abstract. The Test Suite Minimization problem in regression testing is a software engineering problem which consists in selecting a set of test cases from a large test suite that satisfies a given condition, like maximizing the coverage and/or minimizing the oracle cost. In this work we use an approach based on SAT solvers to find optimal solutions for the Test Suite Minimization Problem. The approach comprises two translations: from the original problem instance into Pseudo-Boolean constraints and then to a propositional Boolean formula. In order to solve a problem, we first translate it into a SAT instance. Then the SAT instance is solved using a state-of-the-art SAT solver. Our main contributions are: we create an encoding for single and multi-objective formulations of the Test Suite Minimization Problem as Pseudo-Boolean constraints and we compute optimal solutions for well-known and highly-used instances of this problem for future reference.

Keywords: Test suite minimization, satisfiability problem, multi-objective optimization

1 Introduction

In the last years the performance of Boolean satisfiability (SAT) solvers has been boosted by the introduction of techniques like clause learning, watched literals, and random restarts [1]. Nowadays it is possible to solve SAT instances up to half million variables[1], covering a search space of roughly 2^{500000}. If we compare this cardinality with the cardinality of several combinatorial optimization problems, the difference is considerable in favor of SAT. Thus, we wonder if we can take advantage of this progress in the SAT community to solve interesting optimization problems. In particular, we wonder if we can use the algorithms and tools developed for the SAT problem to find optimal solutions in $\mathcal{NP}$-hard Software Engineering optimization problems, for which metaheuristic techniques are being used at the moment. The answer is yes, and even the SAT community itself has explored other applications of SAT solvers to problems that arise in model checking, planning, and test-pattern generation, among others [2].

[1] The reader can visit http://www.satcompetition.org/ for details.

The main challenge to solve combinatorial optimization problems using SAT solvers is the translation of the target problem to a Boolean Propositional formula. Unfortunately, few problems have an obvious representation as a propositional formula. To overcome this, a common technique is to introduce an intermediate representation for the original problem closer to a Boolean formula. One kind of intermediate representation are Pseudo-Boolean (PB) constraints, which are closely related to SAT. PB constraints provide big expressive power and could be translated to SAT in an automatic way [3].

In this work, we present an approach to solve two variants of the Test Suite Minimization Problem (TSMP) [4] up to optimality using SAT solvers. This is done by modelling TSMP instances as a set of Pseudo-Boolean constraints that are later translated to SAT instances. With the help of a SAT solver the instances are solved and the resulting variable assignment provides an optimal solution for TSMP. Hsu and Orso [5] have tackled this problem in a manner closely related to us, however our contribution over theirs is twofold: we provide the optimal solutions for instances from SIR [6,7], and we apply the approach to a multi-objective formulation of TSMP, obtaining the *Pareto Front* (and a *Pareto optimal set*). A similar approach based on Integer linear programming has been used by Zhang *et al.* for the Time-Aware Test-Case Prioritization [8]. Test-Case prioritisation is a problem related to the TSMP, in which the goal is to find an optimal order in which to execute test cases.

The remainder of this article is structured as follows. In Section 2 we introduce background concepts of SAT solvers and Pseudo-Boolean constraints. In Section 3 we introduce the TSMP and two formulations for this problem: single and multi-objective formulations. In Section 4 we present the application of the proposed approach to the TSMP. Section 5 shows experimental results applying the proposed approach for a set of open well-known instances under the single and multi-objective formulations. Finally with Section 7 we conclude the paper.

2 Background

The Boolean Satisfiability problem (SAT) consists in determining if there exists a Boolean variable assignment that makes *true* a given a propositional Boolean formula. SAT was the first decision problem shown to be $\mathcal{NP}$-Complete [9] and is one of the most important and extensively studied problems, since any other $\mathcal{NP}$ decision problem can be translated into SAT in polynomial time. Thus, if there exists a polynomial time algorithm to solve the SAT problem, then $\mathcal{P} = \mathcal{NP}$. This would answer one of the more important questions in computer science and would bring a great revolution in complexity theory. The algorithms known to solve this problem have complexity $O(2^n)$ in the worst case.

The propositional Boolean formulas are frequently expressed in *Conjunctive Normal Form* (CNF) when they are used as input for the SAT solvers. That is, a Boolean formula is in this case a conjunction of *clauses*, each one consisting in a disjunction of literals (variables negated or not). Let us denote with x_i the Boolean variables for $1 \leq i \leq n$, that is, $x_i \in \{true, false\}$. A *clause* C_j is a

Boolean formula of the form $C_j = x_{j_1} \vee x_{j_2} \vee \ldots \vee x_{j_k} \vee \neg x_{j_{k+1}} \vee \neg x_{j_{k+2}} \vee \ldots \vee \neg x_{j_{k+l}}$. A propositional formula in CNF takes the form $F = \bigwedge_{j=1}^{m} C_j$. When a Boolean formula is expressed in CNF, a solution to a SAT instance consists in an assignment which satisfies all the clauses.

2.1 SAT Solvers

In 1962, Davis, Longemann and Loveland [10] presented a backtracking algorithm based on a systematic search which is the base of current SAT solvers. A backtracking algorithm works by selecting at each step a variable and a Boolean value for branching. In each branching step either *true* or *false* can be assigned to a variable. Then, the logical consequences of each branching are evaluated. Each time a clause becomes unsatisfiable, a backtrack is performed. The backtrack corresponds to undoing branching until a variable is reached for which only one possible Boolean value has been explored. These steps are repeated until the root is reached.

The current state-of-the-art SAT solvers, commonly named *Conflict Driven Clause Learning* (CDCL) solvers, introduce improvements over the described backtracking algorithm. Some of these improvements are:

- **Clause learning:** consists in identifying conflicts between assignments and adding clauses that express these conflicts [11].
- **Non-chronological backtracking:** when a conflict occurs, allows to backtrack to specific decision levels [12].
- **Variable (value) selection heuristic:** establishes rules for determining which variable should be selected and which value the variable should take [13].
- **Random restarts:** allows the restart of the search from scratch. Usually performed as a function of the number of backtracks [14].

These are some of the main techniques included in current CDCL solvers. The interested reader can deepen on these techniques in [1] (chapter 4). In this paper we use the SAT solver MiniSat [15], which is a CDCL solver that includes the techniques described above. MiniSat was designed to be easily extensible, is implemented in C++ (the original source code is under 600 lines) and has been awarded in several categories of the SAT Competition.

2.2 Optimization Problems and Pseudo-Boolean Constraints

SAT is a decision problem, that is, it answers a question (the Boolean formula) with a yes/no answer (satisfiable or unsatisfiable). However, we are interested in optimization problems, in which the goal is to minimize or maximize an objective function. Thus, we need to transform the optimization problem into one or several decision problems that can be translated into a Boolean formula. Let us denote with $f : X \to \mathbb{Z}$ the objective function of the optimization problem[2] and

[2] We focus on integer functions but this is not a hard constraint in practice since floating point numbers in the computers have a finite representation and could be represented with integer numbers.

let us suppose without loss of generality that we want to find a solution $x^* \in X$ that minimizes[3] f, that is, $f(x^*) \leq f(x)$ for all the solutions $x \in X$. This optimization problem can be transformed in a series of decision problems in which the objective is to find a solution $y \in X$ for which the constraint $f(y) \leq B$ holds, where $B \in \mathbb{Z}$ takes different integer values. This series of decision problems can be used to find the optimal (minimal) solution of the optimization problem. The procedure could be as follows. We start with a value of B low enough for the constraint to be unsatisfiabe. We solve the decision problem to check that it is unsatisfiable. Then, we enter a loop in which the value of B is increased and the constraint is checked again. The loop is repeated until the result is satisfiable. Once the loop finishes, the value of B is the minimal value of f in the search space and the solution to the decision problem is an optimal solution of the optimization problem.

If the optimization problem has several objective functions $f_1, f_2, \ldots, f_m$ to minimize, we need one constraint for each objective function:

$$\begin{aligned} f_1(y) &\leq B_1 \\ f_2(y) &\leq B_2 \\ &\vdots \\ f_m(y) &\leq B_m \end{aligned}$$

In order to use SAT solvers to solve optimization problems, we still need to translate the constraints $f(y) \leq B$ to Boolean formulas. To this aim the concept of *Pseudo-Boolean constraint* plays a main role. A Pseudo-Boolean (PB) constraint is an inequality on a linear combination of Boolean variables:

$$\sum_{i=1}^{n} a_i x_i \odot B \tag{1}$$

where $\odot \in \{<, \leq, =, \neq, >, \geq\}$, $a_i, B \in \mathbb{Z}$, and $x_i \in \{0, 1\}$. A PB constraint is said to be *satisfied* under an assignment if the sum of the coefficients a_i for which $x_i = 1$ satisfies the relational operator $\odot$ with respect to B.

PB constraints can be translated into SAT instances. The simplest approaches translate the PB constraint to an equivalent Boolean formula with the same variables. The main drawback of these approaches is that the number of clauses generated grows exponentially with respect to the variables. In practice, it is common to use one of the following methods for the translation: network of adders, binary decision diagrams and network of sorters [1] (chapter 22). All of these approaches introduce additional variables to generate a formula which is semantically equivalent to the original PB constraint. Although the translation of a non-trivial PB constraint to a set of clauses with some of these methods have also an exponential complexity in the worst case, in practice it is not common to have exponential complexity [3] and the translation can be done in a reasonable time.

[3] If the optimization problem consists in maximizing f, we can formulate the problem as the minimization of $-f$.

The PB constraints make easier the translation of combinatorial optimization problems into a SAT instance, since we can use PB constraints as an intermediate step in the translation. This step from the original problem formulation to the set of PB constraints requires human intervention. It is desirable to model the problem using a low number of Boolean variables and PB constraints in order to avoid an uncontrolled increase of the search space. In Section 4 we detail the translation to PB constraints of the Test Suite Minimization Problem.

2.3 Multi-objective Optimization

A general multi-objective optimization problem (MOP) [16] can be formally defined as follows (we assume minimization without loss of generality).

Definition 1 (MOP). *Find a vector* $\boldsymbol{x}^* = (x_1^*, x_2^*, \ldots, x_n^*)$ *which satisfies the* m *inequality constraints* $g_i(\boldsymbol{x}) \geq 0, i = 1, 2, \ldots, m$*, the* p *equality constraints* $h_i(\boldsymbol{x}) = 0, i = 1, 2, \ldots, p$*, and minimizes the vector function* $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), \ldots, f_k(\boldsymbol{x}))$*, where* $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ *is the vector of decision variables.*

The set of all values satisfying the constraints defines the *feasible region* Ω and any point $\boldsymbol{x} \in \Omega$ is a *feasible solution*. It is common in MOPs that not all the objective functions can be simultaneously minimized, there are some conflicts between them. This means that decreasing the value of one objective function implies increasing the value of another one. For this reason, the goal of multi-objective search algorithms is not to find an optimal solution, but a set of *non-dominated solutions* which form the so-called *Pareto optimal set*. We formally define these concepts in the following.

Definition 2 (Pareto Optimality). *A point* $\boldsymbol{x}^* \in \Omega$ *is Pareto optimal if for every* $\boldsymbol{x} \in \Omega$ *and* $I = \{1, 2, \ldots, k\}$ *either* $\forall_{i \in I} f_i(\boldsymbol{x}) = f_i(\boldsymbol{x}^*)$ *or there is at least one* $i \in I$ *such that* $f_i(\boldsymbol{x}) > f_i(\boldsymbol{x}^*)$.

This definition states that $\boldsymbol{x}^*$ is Pareto optimal if no feasible vector $\boldsymbol{x}$ exists which would improve one objective without causing a simultaneous worsening in at least another objective. Other important definitions associated with Pareto optimality are the following:

Definition 3 (Pareto Dominance). *A vector* $\boldsymbol{u} = (u_1, \ldots, u_k)$ *is said to dominate* $\boldsymbol{v}$*=*$(v_1, \ldots, v_k)$ *(denoted by* $\boldsymbol{u} \preccurlyeq \boldsymbol{v}$*) if and only if* $\boldsymbol{u}$ *is partially smaller than* $\boldsymbol{v}$*, i.e.,* $\forall i \in I,\ u_i \leq v_i \ \wedge\ \exists i \in I :\ u_i < v_i$.

Definition 4 (Pareto optimal set). *For a given MOP* $\boldsymbol{f}(\boldsymbol{x})$*, the Pareto optimal set is defined as* $\mathcal{P}^* = \{\boldsymbol{x} \in \Omega | \neg \exists \boldsymbol{x}' \in \Omega, \boldsymbol{f}(\boldsymbol{x}') \preccurlyeq \boldsymbol{f}(\boldsymbol{x})\}$.

Definition 5 (Pareto Front). *For a given MOP* $\boldsymbol{f}(\boldsymbol{x})$ *and its Pareto optimal set* $\mathcal{P}^*$*, the Pareto front is defined as* $\mathcal{PF}^* = \{\boldsymbol{f}(\boldsymbol{x}) | \boldsymbol{x} \in \mathcal{P}^*\}$.

Obtaining the Pareto optimal set and the Pareto front of a MOP are the main goals of multi-objective optimization.

3 Test Suite Minimization Problem

When a piece of software is modified, the new software is tested using some previous test cases in order to check if new errors were introduced. This check is known as regression testing. One problem related to regression testing is the Test Suite Minimization Problem (TSMP). This problem is equivalent to the Minimal Hitting Set Problem which is $\mathcal{NP}$-hard [17]. Let $\mathcal{T} = \{t_1, t_2, \cdots, t_n\}$ be a set of tests for a program where the cost of running test t_i is c_i and let $\mathcal{E} = \{e_1, e_2, \cdots, e_m\}$ be a set of elements of the program that we want to cover with the tests. After running all the tests $\mathcal{T}$ we find that each test can cover several program elements. This information is stored in a matrix $\mathbf{M} = [m_{ij}]$ of dimension $n \times m$ that is defined as:

$$m_{ij} = \begin{cases} 1 & \text{if element } e_j \text{ is covered by test } t_i \\ 0 & \text{otherwise} \end{cases}$$

The single-objective version of this problem consists in finding a subset of tests $X \subseteq \mathcal{T}$ with minimum cost covering all the program elements. In formal terms:

$$minimize \quad cost(X) = \sum_{\substack{i=1 \\ t_i \in X}}^{n} c_i \tag{2}$$

subject to:

$\forall e_j \in \mathcal{E}, \exists t_i \in X$ such that element e_j is covered by test t_i, that is, $m_{ij} = 1$.

The multi-objective version of the TSMP does not impose the constraint of full coverage, but it defines the coverage as the second objective to optimize, leading to a bi-objective problem. In short, the bi-objective TSMP consists in finding a subset of tests $X \subseteq \mathcal{T}$ having minimum cost and maximum coverage. Formally:

$$minimize \quad cost(X) = \sum_{\substack{i=1 \\ t_i \in X}}^{n} c_i \tag{3}$$

$$maximize \quad cov(X) = |\{e_j \in \mathcal{E} | \exists t_i \in X \text{ with } m_{ij} = 1\}| \tag{4}$$

There is no constraint in this bi-objective formulation. We should notice here that solving the bi-objective version (2-obj in short) of TSMP implies solving the single-objective version (1-obj). In effect, let us suppose that we solve an instance of the 2-obj TSMP, then a solution for the related 1-obj TSMP is just the set $X \subseteq \mathcal{T}$ with $cov(X) = |\mathcal{E}|$ in the Pareto optimal set, if such a solution exists. If there is no solution of 2-obj TSMP with $cov(X) = |\mathcal{E}|$, then the related 1-obj TSMP is not solvable.

4 Solving TSMP Instances using PB Constraints

In this section, we will present the proposed approach for solving the TSMP using SAT solvers. First, we detail how the two versions of TSMP can be translated into a set of PB constraints and then we present the algorithms used to solve both versions of TSMP with the help of the SAT solvers.

4.1 Translating the TSMP

The single-objective formulation of TSMP is a particular case of the bi-objective formulation. Then, we can translate the 2-obj TSMP into a set of PB constraints and then infer the translation of the 1-obj TSMP as a especial case.

Let us introduce n binary variables $t_i \in \{0, 1\}$: one for each test case in $\mathcal{T}$. If $t_i = 1$ then the corresponding test case is included in the solution and if $t_i = 0$ the test case is not included. We also introduce m binary variables $e_j \in \{0, 1\}$: one for each program element to cover. If $e_j = 1$ then the corresponding element is covered by one of the selected test cases and if $e_j = 0$ the element is not covered by a selected test case.

The values of the e_j variables are not independent of the t_i variables. A given variable e_j must be 1 if and only if there exists a t_i variable for which $m_{ij} = 1$ and $t_i = 1$. The dependence between both sets of variables can be written with the following $2m$ PB constraints:

$$e_j \leq \sum_{i=1}^{n} m_{ij} t_i \leq n \cdot e_j \qquad 1 \leq j \leq m. \tag{5}$$

We can see that if the sum in the middle is zero (no test is covering the element e_j) then the variable $e_j = 0$. However, if the sum is greater than zero $e_j = 1$. Now we need to introduce a constraint related to each objective function in order to transform the optimization problem in a decision problem, as we described in Section 2.2. These constraints are:

$$\sum_{i=1}^{n} c_i t_i \leq B, \tag{6}$$

$$\sum_{j=1}^{m} e_j \geq P, \tag{7}$$

where $B \in \mathbb{Z}$ is the maximum allowed cost and $P \in \{0, 1, \ldots, m\}$, is the minimum coverage level. We required a total of $n + m$ binary variables and $2m + 2$ PB constraints for the 2-obj TSMP.

For the 1-obj TSMP the formulation is simpler. This is a especial case of the 2-obj formulation in which $P = m$. If we include this new constraint in (7) we have $e_j = 1$ for all $1 \leq j \leq m$. Then we don't need the e_j variables anymore because they are constants. Including these constants in (5) we have:

$$1 \leq \sum_{i=1}^{n} m_{ij} t_i \leq n \qquad 1 \leq j \leq m, \tag{8}$$

which is equivalent to:

$$\sum_{i=1}^{n} m_{ij} t_i \geq 1 \qquad 1 \leq j \leq m, \tag{9}$$

since the sum is always less than or equal to n. Thus, for the 1-obj TSMP the PB constraints are (8) and (9).

4.2 Translation example

In this section we show through a small example how to model with PB constraints an instance of the TSMP according to the methodology above described. Let $\mathcal{T} = \{t_1, t_2, t_3, t_4, t_5, t_6\}$, $\mathcal{E} = \{e_1, e_2, e_3, e_4\}$ and $\mathbf{M}$:

	e_1	e_2	e_3	e_4
t_1	1	0	1	0
t_2	1	1	0	0
t_3	0	0	1	0
t_4	1	0	0	0
t_5	1	0	0	1
t_6	0	1	1	0

If we want to solve the 2-obj TSMP we need to instantiate Eqs. (5), (6) and (7). The result is:

$$e_1 \leq t_1 + t_2 + t_4 + t_5 \leq 4e_1 \tag{10}$$
$$e_2 \leq t_2 + t_6 \leq 4e_2 \tag{11}$$
$$e_3 \leq t_1 + t_3 + t_6 \leq 4e_3 \tag{12}$$
$$e_4 \leq t_5 \leq 4e_4 \tag{13}$$

$$t_1 + t_2 + t_3 + t_4 + t_5 + t_6 \leq B \tag{14}$$
$$e_1 + e_2 + e_3 + e_4 \geq P \tag{15}$$

where $P, B \in \mathbb{N}$.

If we are otherwise interested in the 1-obj version the formulation is simpler:

$$t_1 + t_2 + t_4 + t_5 \geq 1 \tag{16}$$
$$t_2 + t_6 \geq 1 \tag{17}$$
$$t_1 + t_3 + t_6 \geq 1 \tag{18}$$
$$t_5 \geq 1 \tag{19}$$

$$t_1 + t_2 + t_3 + t_4 + t_5 + t_6 \leq B \tag{20}$$

4.3 Algorithms

This section describes the procedures used to find the optimal solutions to the single- and multi-objective formulation of TSMP. Algorithm 1 shows the steps needed to find the optimal solution in the single-objective formulation. We assume, without loss of generality, that full coverage can be reached. If this is not the case we can just remove from $\mathcal{E}$ the program elements that are not covered by any test case.

Algorithm 1 Procedure to compute the optimal solution for 1-obj TSMP

Input: TSMP matrix $\mathbf{M}$
Output: optimal solution $\mathbf{S}^*$
1: $B \leftarrow 1$
2: *result* $\leftarrow$ false
3: **while** not(*result*) **do**
4: Translate $(\mathbf{M}, B)$ into a set of PB constraints: Eqs. (8) and (9)
5: Transform the set of PB constraints into a SAT instance I
6: Run the SAT solver with I as input
7: **if** SAT solver found solution **then**
8: $\mathbf{S}^* \leftarrow$ assignment found
9: *result* $\leftarrow$ true
10: **else**
11: $B \leftarrow B + 1$
12: **end if**
13: **end while**

We can observe that the pseudocode in Algorithm 1 follows the description we introduced in Section 2.2 to solve an optimization problem using SAT solvers. In this case the procedure is adapted to solve the 1-obj TSMP. When the algorithm ends, the value of B is the minimal number of tests required to get full coverage. In the algorithm, the value of B is increased in 1 unit in each iteration (line 11). However, it is possible to use a search strategy based on a binary search in the interval $[1, n]$ for the B value.

In Algorithm 2 we show the procedure used to find a Pareto optimal set for the 2-obj TSMP. In this case the initial value of B will be the one found by the Algorithm 1, which is run in line 2. Each iteration of the loop in line 6 of Algorithm 2 can be seen as a modification of the value P in Eq. (6). This way the algorithm computes the maximum number of elements covered by B test cases. The value of B is decreased in each iteration of the external loop in order to explore the complete Pareto front. Again a binary search could be applied to P or B in order to accelerate the search.

Algorithm 2 Procedure to compute the Pareto optimal set for the 2-obj TSMP

Input: TSMP matrix $\mathbf{M}$
Output: Pareto optimal set

```
1: Pareto optimal set = ∅
2: Run Algorithm 1
3: P = |E|
4: while B ≥ 1 do
5:     found ← false
6:     while (P ≥ 1) and (not(found)) do
7:         Translate (M, B, P) into a set of PB constraints: Eqs. (5), (6) and (7)
8:         Transform the set of PB constraints into a SAT instance I
9:         Run the SAT solver with I as input
10:        if SAT solver found solution then
11:            Add the assignment found to the Pareto optimal set
12:            found ← true
13:        else
14:            P ← P − 1
15:        end if
16:    end while
17:    B ← B − 1
18: end while
```

5 Experimental Results

We performed an experiment to check our approach using the programs from the Siemens suite [6] available at SIR[4] (Software-artifact Infrastructure Repository). The Siemens programs perform a variety of tasks: `printtokens` and `printtokens2` are lexical analyzers, `tcas` is an aircraft collision avoidance system, `schedule` and `schedule2` are priority schedulers, `totinfo` computes statistics given input data, and `replace` performs pattern matching and substitution. The coverage matrix $\mathbf{M}$ can be obtained from the data in the SIR. For the cost values c_i we considered that all the costs are 1: $c_i = 1$ for $1 \leq i \leq n$. As a consequence, the cost function of a set of test cases is just the number of test cases. We implemented Algorithms 1 and 2 as shell scripts in Linux and we used MiniSat+ [3] as Pseudo-Boolean solver. MiniSat+ translates PB constraints into Boolean formulas (in CNF), and uses MiniSat [15] as SAT solver engine.

In a second experiment we transformed the instances into equivalent ones with fewer test cases. We can do this because we are considering that all test cases have the same cost. Under this assumption we can remove any test case for which there is another test case covering at least all the elements covered by the first one. In formal terms, if test case t_i covers program elements $E_i \subseteq \mathcal{E}$ and test case t_h covers program elements $E_h \subseteq \mathcal{E}$ where $E_i \subseteq E_h$, then we remove t_i from the original test suite and the Pareto front of the instance does not change, because any solution having test case t_i cannot get worse after replacing t_i by t_h.

[4] `http://sir.unl.edu/portal/index.php`

The result is an instance with fewer test cases but having the same Pareto front. These transformed instances were solved using Algorithm 2. Table 1 shows the size of the test suites with and without the reduction for each program. We can observe a really great reduction in the number of test cases when the previous approach is used.

Table 1: Details of the instances used in the experiments

Instance	Original Size	Reduced Size	Elements to cover
`printtokens`	4130	40	195
`printtokens2`	4115	28	192
`replace`	5542	215	208
`schedule`	2650	4	126
`schedule2`	2710	13	119
`tcas`	1608	5	54
`totinfo`	1052	21	117

In Table 2 we present the Pareto optimal set and the Pareto front for the instances described above. The columns "Tests" and "Elements" correspond to the functions *cost* and *cov* of the 2-obj TSMP. The column "Coverage" is the number of covered elements divided by the total number of elements. The optimal solution for the 1-obj TSMP can be found in the lines with 100% coverage, as explained in Section 3. It is not common to show the Pareto optimal set or the Pareto front in numbers in the multi-objective literature because only approximate Pareto fronts can be obtained for $\mathcal{NP}$-hard problems. However, in this case we obtain the exact Pareto fronts and optimal sets, so we think that this information could be useful for future reference. Figure 1 shows the Pareto front for all the instances of Table 1: they present the same information as Table 2 in a graphical way. The information provided in the tables and the figures is very useful for the tester, knowing beforehand which are the most important test cases and giving the possibility to make a decision taking into account the number of tests necessary to assure a particular coverage level or vice versa.

We show in Table 3 the running time of Algorithm 2, which includes the execution of Algorithm 1. The experiments were performed on a Laptop with an Intel CORE i7 running Ubuntu Linux 11.04. Since the underlying algorithm is deterministic the running time is an (almost) deterministic variable. The only source of randomness for the SAT solver comes from limited random restarts and the application of variable selection heuristics. Additionally, we compared the running time of our approach with the performance of two heuristic algorithms: a local search (LS) algorithm and a genetic algorithm (GA) for the 1-obj formulation of the TSMP. The LS algorithm is based on an iterative best improvement process and the GA is a steady-state GA with 10 individuals in the population, binary tournament selection, bit-flip mutation with probability $p = 0.01$ of flipping a bit, one-point crossover and elitist replacement. The stopping condition is

Table 2: Pareto optimal set and Front for the instances of SIR.

Instance	Elements	Tests	Coverage	Solution
`printtokens`	195	5	100%	$(t_{2222}, t_{2375}, t_{3438}, t_{4100}, t_{4101})$
	194	4	99.48%	$(t_{1908}, t_{2375}, t_{4099}, t_{4101})$
	192	3	98.46%	$(t_{1658}, t_{2363}, t_{4072})$
	190	2	97.43%	(t_{1658}, t_{3669})
	186	1	95.38%	(t_{2597})
`printtokens2`	192	4	100%	$(t_{2521}, t_{2526}, t_{4085}, t_{4088})$
	190	3	98.95%	$(t_{457}, t_{3717}, t_{4098})$
	188	2	97.91%	(t_{2190}, t_{3282})
	184	1	95.83%	(t_{3717})
`replace`	208	8	100%	$(t_{306}, t_{410}, t_{653}, t_{1279}, t_{1301}, t_{3134}, t_{4057}, t_{4328})$
	207	7	99.51%	$(t_{309}, t_{358}, t_{653}, t_{776}, t_{1279}, t_{1795}, t_{3248})$
	206	6	99.03%	$(t_{275}, t_{290}, t_{1279}, t_{1938}, t_{2723}, t_{2785})$
	205	5	98.55%	$(t_{426}, t_{1279}, t_{1898}, t_{2875}, t_{3324})$
	203	4	97.59%	$(t_{298}, t_{653}, t_{3324}, t_{5054})$
	200	3	96.15%	$(t_{2723}, t_{2901}, t_{3324})$
	195	2	93.75%	(t_{358}, t_{5387})
	187	1	89.90%	(t_{358})
`schedule`	126	3	100%	$(t_{1403}, t_{1559}, t_{1564})$
	124	2	98.41%	(t_{1570}, t_{1595})
	122	1	96.82%	(t_{1572})
`schedule2`	119	4	100%	$(t_{2226}, t_{2458}, t_{2462}, t_{2681})$
	118	3	99.15%	$(t_{101}, t_{1406}, t_{2516})$
	117	2	98.31%	(t_{2461}, t_{2710})
	116	1	97.47%	(t_{1584})
`tcas`	54	4	100%	$(t_{5}, t_{1191}, t_{1229}, t_{1608})$
	53	3	98.14%	$(t_{13}, t_{25}, t_{1581})$
	50	2	92.59%	(t_{72}, t_{1584})
	44	1	81.48%	(t_{217})
`totinfo`	117	5	100%	$(t_{62}, t_{118}, t_{218}, t_{1000}, t_{1038})$
	115	4	98.29%	$(t_{62}, t_{118}, t_{913}, t_{1016})$
	113	3	96.58%	$(t_{65}, t_{216}, t_{913})$
	111	2	94.87%	(t_{65}, t_{919})
	110	1	94.01%	(t_{179})

to equal the running time of the SAT-based method for each reduced instance. For the two heuristic algorithms we show the average coverage and number of test cases over 30 independent runs.

Regarding the computational time, we observe that all the instances can be solved in much less time using the reduction. The speed up for the SAT-based approach ranges from more than 200 for `tcas` to more than 2000 for `printtokens2`. All the instances can be solved in around 2 seconds with the exception of `replace`, which requires almost 6 minutes. In the case of the heuristic algorithms, we observe that LS reaches full coverage in all the instances and

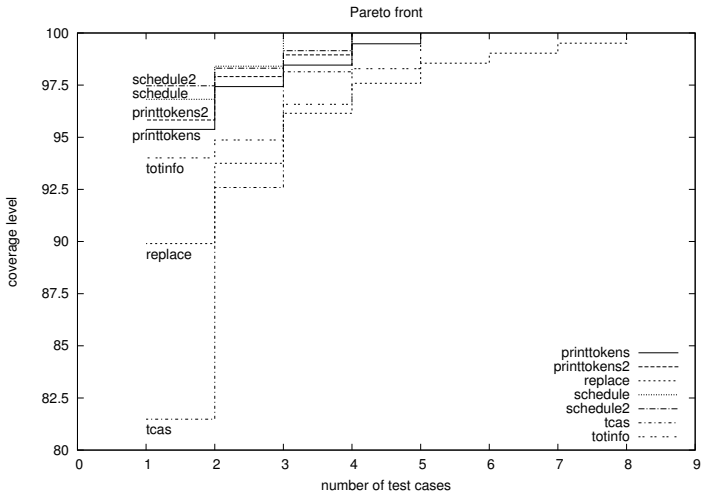

Fig. 1: Pareto front for the SIR instances

Table 3: Information about clauses-to-variables ratio, computation time of Algorithm 2, average coverage and number of test cases for the two heuristic algorithms for the instances from SIR.

Instance	Ratio	Algorithm 2		Local Search		Genetic Algorithm	
		Original (s)	Reduced (s)	Avg. Cov.	Avg. Tests	Avg. Cov.	Avg. Tests
`printtokens`	4.61	3400.74	2.17	100.00%	6.00	99.06%	5.16
`printtokens2`	4.61	3370.44	1.43	100.00%	4.60	99.23%	3.56
`replace`	4.62	1469272.00	345.62	100.00%	10.16	99.15%	15.46
`schedule`	2.19	492.38	0.24	100.00%	3.00	99.84%	2.90
`schedule2`	4.61	195.55	0.27	100.00%	4.00	99.58%	3.70
`tcas`	4.61	73.44	0.33	100.00%	4.00	95.80%	3.23
`totinfo`	4.53	181823.50	0.96	100.00%	5.00	98.89%	5.13

independent runs. However, the required number of test cases is non-optimal in `printtokens`, `printtokens2` and `replace`. LS obtains optimal solutions in the rest of the programs. However, we should recall here that LS cannot ensure that the result is an optimal solution, as the SAT-based approach does. In the case of GA, it is not able to reach full coverage in any program.

It is interesting to remark that almost all the resulting SAT instances obtained from the translation are in the phase transition of SAT problems except the one for `schedule`. It has been shown experimentally that most of the instances where the ratio of clauses-to-variables is approximately equal to 4.3 are the hardest to be solved [18].

6 Threats to validity

We identified two major threats to validity in our approach. The first one is related to the inherent difficulty of the SAT problem for which there is no known efficient algorithm to solve it. This implies that there is a limit in the number of test cases that can be addressed with our method. For example, in the experiments we observed a high execution time to solve two of the seven instances without the reduction: `replace` and `totinfo`. Further experiments with more and larger instances could show what are the actual limitation of our approach in real-world software.

The second one concerns the complexity of the translation from the PB constraints to SAT clauses. This complexity is exponential in the worst-case but not in practice. In our experiments the time required for this translation was very small. We think this is a consequence of having unitary cost for the tests. If we consider non-unitary cost for the tests, the translation could require a non-negligible amount of time. Furthermore, the resulting SAT instance could be more difficult to solve in this case, making the approach impractical. To study these issues we need more experiments using larger instances and non-unitary costs for the tests. These costs could be based on the tests execution time.

7 Conclusion and Future Work

In this work we show an approach to optimally solve the TSMP. This approach comprises two translations to obtain a SAT instance which is solved by a state-of-the-art SAT solver. The power of current SAT solvers give us the possibility of solving to optimality TSMP instances that were previously solved in an approximate way using metaheuristic algorithms. With the help of MiniSat+ we solved well-known and highly-used instances of a single- and a bi-objective formulation of the TSMP problem. Most of the instances were solved in less than 2 seconds and all of them required less than 6 minutes.

The approach presented here to solve the TSMP problem can be easily extended to other hard problems in Search-Based Software Engineering and other domains. As future work we plan to consider different cost coefficients for the different test cases in the TSMP.

Acknowledgements.

This research has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contract TIN2011-28194 (RoadME project) and the Andalusian Government under contract P07-TIC-03044 (DIRICOM project).

References

1. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, The Netherlands, The Netherlands (2009)

2. Marques-Silva, J.: Practical applications of boolean satisfiability. In: Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on. (may 2008) 74 –80
3. Eén, N., Sörensson, N.: Translating Pseudo-Boolean Constraints into SAT. Journal on Satisfiability, Boolean Modeling and Computation **2**(1-4) (2006) 1–26
4. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability (2010) doi: 10.1002/stvr.430.
5. Hsu, H.Y., Orso, A.: MINTS: A general framework and tool for supporting test-suite minimization. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. (may 2009) 419 –429
6. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: Proceedings of the 16th international conference on Software engineering. ICSE '94, Los Alamitos, CA, USA, IEEE Computer Society Press (1994) 191–200
7. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering **10** (2005) 405–435
8. Zhang, L., Hou, S.S., Guo, C., Xie, T., Mei, H.: Time-aware test-case prioritization using integer linear programming. In: Proceedings of the eighteenth international symposium on Software testing and analysis. ISSTA '09, New York, NY, USA, ACM (2009) 213–224
9. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. STOC '71, New York, NY, USA, ACM (1971) 151–158
10. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**(7) (July 1962) 394–397
11. Silva, J.P.M., Sakallah, K.A.: Grasp: A new search algorithm for satisfiability. In: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design, Washington, DC, USA, IEEE Computer Society (1996) 220–227
12. Stallman, R.M., Sussman, G.J.: Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. Artificial Intelligence **9**(2) (1977) 135 – 196
13. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th annual Design Automation Conference. DAC '01, New York, NY, USA, ACM (2001) 530–535
14. Gomes, C., Selman, B., Crato, N.: Heavy-tailed distributions in combinatorial search. In Smolka, G., ed.: Principles and Practice of Constraint Programming-CP97. Volume 1330 of LNCS. Springer Berlin / Heidelberg (1997) 121–135
15. Eén, N., Sörensson, N.: An extensible SAT-solver. In Giunchiglia, E., Tacchella, A., eds.: Theory and Applications of Satisfiability Testing. Volume 2919 of LNCS. Springer Berlin / Heidelberg (2004) 333–336
16. Deb, K.: Multi-objective optimization using evolutionary algorithms. John Wiley & Sons (2001)
17. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., New York, NY, USA (1979)
18. Mitchell, D., Selman, B., Levesque, H.: Hard and easy distributions of SAT problems. In: Proceedings of the tenth national conference on Artificial intelligence. AAAI'92, AAAI Press (1992) 459–465